UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

Práctica 1
MPI

Carlos Gómez Fernández

Asignatura: Diseño de Infraestructura de Red

Titulación: Grado en Ingeniería Informática

Fecha: 14 de Abril de 2021

ÍNDICE.

1. RED TOROIDE.

1.1.ENUNCIADO DEL PROBLEMA.

Dado un archivo con nombre *datos.dat*, cuyo contenido es una lista de valores separados por comas, el programa realizará las siguientes tareas:

El proceso con *rank 0* distribuirá a cada uno de los nodos que conforman la red Toroide de lado *L*, los *LxL* números reales que contendrá el archivo *datos.dat*. En caso de que no se hayan lanzado suficientes elementos de proceso para los datos del programa, éste emitirá un error y todos los procesos finalizarán.

En caso de que todos los procesos reciban su correspondiente elemento, comenzará la ejecución normal del programa.

Se pide calcular el menor elemento de toda la red, tras lo cual, el elemento de proceso con *rank 0* mostrará en su salida estándar el valor obtenido. La complejidad del algoritmo no superará $O(\sqrt{n})$, siendo *n*, el número de elementos de la red.

1.2.PLANTEAMIENTO DE LA SOLUCIÓN.

1.2.1. FUNDAMENTOS TEÓRICOS.

Previo a plantear la solución del problema, se deben tener en cuenta las características teóricas de una red Toroide, cuyo tamaño se define por el tamaño del lado de la misma, es decir, por la cantidad de nodos que se encuentran en cualquier lado de la red, ya que, se corresponde con un red simétrica, por lo que la cantidad total de nodos que conforman la red será *N*, siendo $N = LxL$.

Cada nodo de una red Toroide se conecta con cuatro nodos adyacentes, correspondientes con los puntos cardinales, disponiendo cada nodo del Toroide de los nodos vecinos *Norte*, *Sur*, *Este* y *Oeste*.

En la red Toroide diseñada y empleada en la resolución del problema la numeración se basará en 0, es decir, los nodos se encontrarán representados en el rango de valores desde 0 hasta *N-1*, o su equivalente, desde 0 hasta *(LxL)-1*.

1.2.2. PLANTEAMIENTO TEÓRICO DE LA SOLUCIÓN.

Para llevar a cabo el planteamiento de la solución, se deberá estudiar el funcionamiento requerido para cada uno de los nodos de la red, pudiendo distinguir entre dos funcionamientos principales, el correspondiente con el nodo o proceso con *rank 0* y el del resto de nodos o procesos.

Entre las funcionalidades asignadas al proceso con *rank 0* se encuentran la del procesamiento de los valores almacenados en el archivo *datos.dat* y su posterior distribución entre el resto de procesos, el control del lanzamiento de procesos en relación con el tamaño de la red, el control del almacenamiento de valores suficientes en el archivo *datos.dat*, respecto al número de procesos lanzados y el control de la correcta ejecución del resto de procesos, para finalmente mostrar el valor requerido.

Entre las funcionalidades asignadas al resto de procesos de la red se encuentran la de la recepción del valor enviado por el proceso con *rank 0*, la de la obtención de los vecinos adyacentes y la del procesamiento y posterior envío del valor mínimo, en cada caso, a sus vecinos.

Para llevar a cabo todas estas funcionalidades, tanto la del proceso con *rank 0* como las del resto de los procesos, se deben tener en cuanta ciertas situaciones en las que se debe detener la ejecución del programa, como cuando el número de procesos lanzados sea distinto a la cantidad total de nodos o procesos que conforman la red y cuando la cantidad de valores almacenados en el archivo *datos.dat* sea distinto al número total de nodos o procesos que conforman la red.

1.3.DISEÑO DEL PROGRAMA.

1.3.1. FUNCIONALIDADES DEL PROCESO CON RANK 0.

Para llevar a cabo la comprobación de que el número de procesos lanzados es igual a la cantidad de nodos o procesos de los que dispone la red creada, se empleará la función *MPI_Comm_size(MPI_COMM_WORLD, &size)*, mediante la cual, se obtiene el número total de procesos lanzados. En caso de que los valores sean diferentes se enviará, mediante la instrucción *MPI_Bcast()*, un mensaje para que el resto de los procesos finalicen su ejecución.

Para llevar a cabo la lectura del fichero y procesar posteriormente los datos almacenados en el mismo, se ha definido la función *obtenerNumeros()*, mediante la cual se obtendrá, tanto el cantidad total de números, como un vector con los números almacenados en el fichero *datos.dat*.

Tras procesar el archivo *datos.dat* y obtener los números almacenados en él, se debe comprobar si la cantidad de números leídos es igual a la cantidad de nodos o procesos que pueden ser lanzados en una red Toroide del tamaño asignado anteriormente. Si este valor es diferente se enviará, mediante la instrucción *MPI_Bcast()*, un mensaje para que el resto de los procesos finalicen su ejecución.

Para llevar a cabo el envío de datos desde el proceso con *rank 0*, hasta el resto de los procesos, se empleará la instrucción *MPI_Send()*, mediante la cual se enviará un número a cada uno de los nodos o procesos que conforman la red, para que todos, incluido el proceso con *rank 0*, dispongan de uno de los números almacenados en el archivo *datos.dat.*

1.3.2. FUNCIONALIDADES DEL RESTO DE PROCESOS.

Para iniciar la ejecución de los procesos se hará uso de la instrucción *MPI_Bcast()*, la cual permite sincronizar los procesos, ya sea finalizándolos, como sucedía anteriormente, o iniciando su ejecución, mediante un *flag*, en este caso denominado *control*, el cual si su valor es *1* inicia la ejecución de los procesos, en cambio si es *0*, la finaliza.

Para llevar a cabo la obtención de los datos se empleará la instrucción *MPI_Recv()*, la cual mantendrá al proceso a la espera hasta la recepción del dato enviado previamente por el proceso con *rank 0*.

Para llevar a cabo la obtención de los vecinos o nodos adyacentes a un nodo, en las posiciones descritas anteriormente, se ha definido la función *obtenerVecinos()*, la cual los obtendrá en función del *rank* de cada nodo y de la cantidad total de nodos de la red. Para obtener la fila en la que se encuentra un nodo concreto basta con dividir su posición (definida por el *rank*) entre el valor del lado de la red, y para obtener la columna en la que se encuentra un nodo concreto basta con hacer el módulo de su posición (definida por el *rank*) entre el valor del lado de la red. El *rank* de los nodos se asignará de izquierda a derecha y de abajo hacia arriba, igual que la enumeración de las filas y las columnas.

Para obtener los vecinos adyacentes se debe tener en cuenta todas las consideraciones posibles, es decir, si es la fila inferior, una interior o la superior, y se desea conocer el vecino *Norte* o *Sur*, y si es la columna de la izquierda, una interior o la de la derecha y se desea conocer el vecino *Este* u *Oeste*, pero sobre todo las posibilidades correspondientes con los casos que necesitan algún tipo de tratamiento especial, como es el caso de un nodo en la fila inferior intentando obtener su vecino correspondiente a la posición *Sur*, un nodo en la fila superior intentando obtener su vecino correspondiente a la posición *Norte*, un nodo en la columna izquierda intentando obtener su vecino correspondiente a la posición *Oeste* y un nodo en la columna derecha intentando obtener su vecino correspondiente a la posición *Este*, las cuales se encuentran resumidas en la siguiente tabla:

Fila	Vecino Norte	Vecino Sur
Inferior (0)	$((fila + 1) * L) + columna$	$((L - 1) * L) + columna$
Interior (1 - ((L-1)-1))	$((fila + 1) * L) + columna$	$((fila - 1) * L) + columna$
Superior (L-1)	$columna$	$((fila - 1) * L) + columna$

Columna	Vecino Este	Vecino Oeste
Izquierda (0)	$(fila * L) + (columna + 1)$	$(fila * L) + (L - 1)$
Interior (1 - ((L-1)-1))	$(fila * L) + (columna + 1)$	$(fila * L) + (columna - 1)$
Derecha (L-1)	$fila * L$	$(fila * L) + (columna - 1)$

Para obtener el valor mínimo de los datos almacenados entre todos los nodos de la red se ha definido la función *obtenerMinimo()*, en la cual, primero se realiza la comunicación entre los nodos adyacentes en las posiciones *Este* y *Oeste*, en cada uno de los cuales se compara si el valor que fue enviado por el proceso con *rank 0*, es mayor o menor que el enviado, mediante la instrucción *MPI_Send()*, por el por el vecino adyacente en la posición *Este*. En el caso de que el primer valor sea menor que el segundo, dicho nodo mantendrá el valor inicial, en cambio, en el caso opuesto, el nodo almacenará el segundo valor, es decir, se quedará siempre con el menor. Tras esto, los nodos de la red serán conocedores del valor mínimo de cada una de las filas de la red, por lo que deberán comunicarse con los nodos adyacentes en las posiciones *Norte* y *Sur*, en cada uno de los cuales se compara si el valor que almacena cada nodo es mayor o menor que el enviado, mediante la instrucción *MPI_Send()*, por el vecino adyacente en la posición *Norte*. En el caso de que el primer valor sea menor que el segundo, dicho nodo mantendrá el valor inicial, en cambio, en el caso opuesto, el nodo almacenará el segundo valor, es decir, se quedará siempre con el menor. Tras esto, todos los nodos de la red serán conocedores del valor mínimo de la red.

Para mostrar el valor mínimo que almacena la red, el nodo o proceso con *rank 0*, imprimirá por pantalla el valor devuelto por la función *obtenerMinimo()*.

2. RED HIPERCUBO.

2.1.ENUNCIADO DEL PROBLEMA.

Dado un archivo con nombre *datos.dat*, cuyo contenido es una lista de valores separados por comas, el programa realizará las siguientes tareas:

El proceso con *rank 0* distribuirá a cada uno de los nodos que conforman la red *Hipercubo* de dimensión *D*, los 2^D números reales que contendrá el archivo *datos.dat*. En caso de que no se hayan lanzado suficientes elementos de proceso para los datos del programa, éste emitirá un error y todos los procesos finalizarán.

En caso de que todos los procesos reciban su correspondiente elemento, comenzará la ejecución normal del programa.

Se pide calcular el mayor elemento de toda la red, tras lo cual, el elemento de proceso con *rank 0* mostrará en su salida estándar el valor obtenido. La complejidad del algoritmo no superará *O(*$\log(n)$*)*, siendo *n*, el número de elementos de la red.

2.2.PLANTEAMIENTO DE LA SOLUCIÓN.

2.2.1. FUNDAMENTOS TEÓRICOS.

Previo a plantear la solución del problema, se deben tener en cuenta las características teóricas de una red Hipercubo, cuyo tamaño se define por el grado o dimensión (D) de los nodos de la misma, es decir, por la cantidad de nodos adyacentes que tiene cada nodo de la red, por lo que la cantidad total de nodos que conforman la red será *N*, siendo $N = 2^D$.

En la red Hipercubo diseñada y empleada en la resolución del problema la numeración se basará en 0, es decir, los nodos se encontrarán representados en el rango de valores desde 0 hasta *N-1*, o su equivalente, desde 0 hasta *(*2^D*)-1*.

2.2.2. PLANTEAMIENTO TEÓRICO DE LA SOLUCIÓN.

Para llevar a cabo el planteamiento de la solución, se deberá estudiar el funcionamiento requerido para cada uno de los nodos de la red, pudiendo distinguir entre dos funcionamientos principales, el correspondiente con el nodo o proceso con *rank 0* y el del resto de nodos o procesos.

Entre las funcionalidades asignadas al proceso con *rank 0* se encuentran la del procesamiento de los valores almacenados en el archivo *datos.dat* y su posterior distribución entre el resto de procesos, el control del lanzamiento de procesos en relación con el tamaño de la red, el control del almacenamiento de valores suficientes en el archivo *datos.dat*, respecto al número de procesos lanzados y el control de la correcta ejecución del resto de procesos, para finalmente mostrar el valor requerido.

Entre las funcionalidades asignadas al resto de procesos de la red se encuentran la de la recepción del valor enviado por el proceso con *rank 0*, la de la obtención de los vecinos adyacentes y la del procesamiento y posterior envío del valor máximo, en cada caso, a sus vecinos.

Para llevar a cabo todas estas funcionalidades, tanto la del proceso con *rank 0* como las del resto de los procesos, se deben tener en cuanta ciertas situaciones en las que se debe detener la ejecución del programa, como cuando el número de procesos lanzados sea distinto a la cantidad total de nodos o procesos que conforman la red y cuando la cantidad de valores almacenados en el archivo *datos.dat* sea distinto al número total de nodos o procesos que conforman la red.

2.3.DISEÑO DEL PROGRAMA.

2.3.1. FUNCIONALIDADES DEL PROCESO CON RANK 0.

Para llevar a cabo la comprobación de que el número de procesos lanzados es igual a la cantidad de nodos o procesos de los que dispone la red creada, se empleará la función *MPI_Comm_size(MPI_COMM_WORLD, &size)*, mediante la cual, se obtiene el número total de procesos lanzados. En caso de que los valores sean diferentes se enviará, mediante la instrucción *MPI_Bcast()*, un mensaje para que el resto de los procesos finalicen su ejecución.

Para llevar a cabo la lectura del fichero y procesar posteriormente los datos almacenados en el mismo, se ha definido la función *obtenerNumeros()*, mediante la cual se obtendrá, tanto el cantidad total de números, como un vector con los números almacenados en el fichero *datos.dat*.

Tras procesar el archivo *datos.dat* y obtener los números almacenados en él, se debe comprobar si la cantidad de números leídos es igual a la cantidad de nodos o procesos que pueden ser lanzados en una red Hipercubo del tamaño asignado anteriormente. Si este valor es diferente se enviará, mediante la instrucción *MPI_Bcast()*, un mensaje para que el resto de los procesos finalicen su ejecución.

Para llevar a cabo el envío de datos desde el proceso con *rank 0*, hasta el resto de los procesos, se empleará la instrucción *MPI_Send()*, mediante la cual se enviará un número a cada uno de los nodos o procesos que conforman la red, para que todos, incluido el proceso con *rank 0*, dispongan de uno de los números almacenados en el archivo *datos.dat.*

2.3.2. FUNCIONALIDADES DEL RESTO DE PROCESOS.

Para iniciar la ejecución de los procesos se hará uso de la instrucción *MPI_Bcast()*, la cual permite sincronizar los procesos, ya sea finalizándolos, como sucedía anteriormente, o iniciando su ejecución, mediante un *flag*, en este caso denominado *control*, el cual si su valor es *1* inicia la ejecución de los procesos, en cambio si es *0*, la finaliza.

Para llevar a cabo la obtención de los datos se empleará la instrucción *MPI_Recv()*, la cual mantendrá al proceso a la espera hasta la recepción del dato enviado previamente por el proceso con *rank 0*.

Para llevar a cabo la obtención de los vecinos o nodos adyacentes a un nodo, se ha definido la función *obtenerVecinos()*, la cual los obtendrá en función del *rank* de cada nodo y de la dimensión de la red, realizando la operación *XOR* entre ambos valores.

Para obtener el valor máximo de los datos almacenados entre todos los nodos de la red se ha definido la función *obtenerMaximo()*, en la cual, un nodo envía el dato que almacena a todos los nodos adyacentes, y recibe, de estos, los valores que almacenan. Cada nodo comprueba si el número recibido es mayor al almacenado previamente, consiguiendo que todos los nodos de la red almacenen el de mayor valor.

Para mostrar el valor máximo que almacena la red, el nodo o proceso con *rank 0*, imprimirá por pantalla el valor devuelto por la función *obtenerMaximo()*.

3. EXPLICACIÓN DEL FLUJO DE DATOS EN LA RED PARA CADA COMANDO MPI.

Los comandos *MPI* utilizados en la resolución del problema han sido *MPI_Bcast()*, *MPI_Send()* y *MPI_Recv()*.

MPI_Bcast() corresponde con una operación colectiva empleada para la multidifusión de datos y ha sido empleada en la solución al problema para le detención de la ejecución de todos los procesos de la red en los casos detallados anteriormente, cuya sintaxis es la siguiente *int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)*, adaptada a la solución del problema sería *MPI_Bcast(&control, 1, MPI_INT, 0, MPI_COMM_WORLD)*.

MPI_Send() corresponde con una primitiva que permite el envío de datos cuya sintaxis es la siguiente *int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*, adaptada a la solución del problema sería *MPI_Send(&minimo, 1, MPI_DOUBLE, vecinoEste, i, MPI_COMM_WORLD)*.

MPI_Recv() corresponde con una primitiva que permite la recepción de datos cuya sintaxis es la siguiente *int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)*, adaptada a la solución del problema sería *MPI_Recv(&bufNumeros, 1, MPI_DOUBLE, vecinoOeste, i, MPI_COMM_WORLD, &status)*.

4. FUENTES DEL PROGRAMA.

4.1.OBTENERNUMEROS.C.

```c
/* GENERACION DE UN ARCHIVO DE DATOS CON NUMEROS SEPARADOS POR COMAS */
/* Carlos Gomez Fernandez */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* Funcion para generar numeros aleatorios en un rango de valores */
double generarNumerosAleatorios(int minimo, int maximo);

int main(int argc, int *argv[]){
    /* Lectura de la entrada estandar en la que se introduce la cantidad de nu-
meros a generar */
    int numerosTotales = atoi(argv[1]);
    /* Apertura del archivo en modo escritura */
    FILE *archivo = fopen("datos.dat", "w");
    /* Creacion de la semilla */
    srand(time(NULL));
    /* Bucle para almacenar en el archivo datos.dat tantos numeros como se ha-
yan introducido por la entrada estandar */
    for (int i=0; i<numerosTotales; i++){
        fprintf(archivo, "%1.2f,", generarNumerosAleatorios(-1000,1000));
    }
}

/* Funcion para generar numeros aleatorios en un rango de valores */
double generarNumerosAleatorios(int minimo, int maximo){
    return (rand() % (maximo - minimo + 1) + minimo)/10.0f;}
```

4.2. TOROIDE.C.

```
/* OBTENCION DEL VALOR MINIMO DE LOS DATOS DISTRIBUIDOS ENTRE LOS NO-
DOS DE UNA RED TOROIDE */
/* Carlos Gomez Fernandez */

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <string.h>

/* Archivo donde se almacenan los numeros a procesar */
#define FICHERO "datos.dat"

/* Funciones empleadas para la resolucion del problema */
int obtenerNumeros(double *numeros);
void obtenerVecinos(int rank, int *vecinoNorte, int *vecinoSur, int *ve-
cinoEste, int *vecinoOeste);
double obtenerMinimo(int rank, int vecinoNorte, int vecinoSur, int ve-
cinoEste, int vecinoOeste, double bufNumeros);

int main(int argc, char *argv[]){
    int rank, size, numerosProcesar, vecinoNorte, vecinoSur, vecinoEste, veci-
noOeste, control = 1;
    double bufNumeros, minimo;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    /* Control para que el rank 0 sea el que lleve a cabo el procesa-
miento y distribucion de los datos */
    if (rank == 0){
        /* Control del lanzamiento de procesos suficientes, acorde con el ta-
mano definido de la red */
        if (L*L != size){
            printf("[ERROR] Para un toroide de lado %d deben ser lanza-
dos un total %d procesos\n", L, L*L);
            /* Finalizacion de la ejecucion de los procesos */
            control = 0;
            MPI_Bcast(&control, 1, MPI_INT, 0, MPI_COMM_WORLD);
        } else {
            /* Vector de numeros a repartir entre los nodos de la red */
            double *numeros = malloc(1024 * sizeof(double));
            /* Obtencion de los numeros almacenados en el archivo datos.dat */
            numerosProcesar = obtenerNumeros(numeros);
```

```
            /* Control de la cantidad de numeros almacenados en el archivo da-
tos.dat, acorde con el tamano definido de la red */
            if(L*L != numerosProcesar){
                printf("[ERROR] Este toroide necesita únicamente %d nume-
ros y dispone de %d numeros en el fichero\n", L*L, numerosProcesar);
                /* Finalizacion de la ejecucion de los procesos */
                control = 0;
                MPI_Bcast(&control, 1, MPI_INT, 0, MPI_COMM_WORLD);
            } else {
                /* Continuar con la ejecucion de los procesos */
                MPI_Bcast(&control, 1, MPI_INT, 0, MPI_COMM_WORLD);
                /* Envio de los numeros a cada uno de los nodos de la red */
                for(int i = 0; i<numerosProcesar; i++){
                    bufNumeros = numeros[i];
                    MPI_Send(&bufNumeros, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
                }
            }
        }
    }
    /* El rank 0 envia un mensaje para continuar con la ejecucion de los proce-
sos, tras el envio de los numeros a cada uno de los nodos de la red */
    MPI_Bcast(&control, 1, MPI_INT, 0, MPI_COMM_WORLD);
    /* Control para la continuacion de la ejecucion de los procesos */
    if (control != 0){
        /* Recepcion de todos los nodos de los numeros envia-
dos por el rank 0 */
        MPI_Recv(&bufNumeros, 1, MPI_DOU-
BLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        /* Cada nodo obtiene sus vecinos */
        obtenerVecinos(rank, &vecinoNorte, &vecinoSur, &vecinoEste, &veci-
noOeste);
        /* Obtencion del valor minimo almacenado en la red */
        minimo = obtenerMinimo(rank, vecinoNorte, vecinoSur, vecinoEste, veci-
noOeste, bufNumeros);
        /* Control para que el rank 0 sea el que muestre el valor mi-
nimo de la red */
        if (rank == 0){
            printf("El menor numero que almacena este toroide es %.1f\n", mi-
nimo);
        }
    }
    /* Finalizacion de la ejecucion */
    MPI_Finalize();
    return 0;
}

/* Funcion para obtener los numeros almacenados en el archivo datos.dat */
int obtenerNumeros (double *numeros){
```

```
    /* Vector auxiliar de char para almacenar los numeros separados por co-
mas */
    char *numerosFichero = malloc(1024 * sizeof(char));
    int *numeroActual;
    int totalNumeros = 0;

    /* Apertura del archivo en modo lectura */
    FILE *archivo = fopen(FICHERO, "r");
    /* Control de la correcta apertura del archivo */
    if (archivo == NULL){
        fprintf(stderr, "[ERROR] Error en la apertura del archivo\n");
        return 0;
    }
    /* Copia de los numeros en un vector auxiliar */
    fscanf(archivo, "%s", numerosFichero);
    /* Cierre del archivo */
    fclose(archivo);
    /* Lectura del primer numero entre comas mediante strtok y transforma-
cion a double con atof */
    numeros[totalNumeros++] = atof(strtok(numerosFichero,","));
    /* Lectura de los numeros entre comas*/
    while ((numeroActual = strtok(NULL,",")) != NULL){
        numeros[totalNumeros++] = atof(numeroActual);
    }

    return totalNumeros;
}

/* Funcion para obtener los vecinos de cada nodo */
void obtenerVecinos(int rank, int *vecinoNorte, int *vecinoSur, int *ve-
cinoEste, int *vecinoOeste){
    int fila, columna;
    /* Las filas se enumeraran de abajo hacia arriba y las columnas de iz-
quierda a derecha */
    fila = rank/L;
    columna = rank%L;
    /* Control para la busqueda del vecino Sur por parte de la fila inferior */
    if (fila == 0){
        /* Parte superior de la red */
        *vecinoSur = ((L-1)*L)+columna;
    } else {
        /* Control para la busqueda del vecino Sur por parte de cual-
quier fila */
        *vecinoSur = ((fila-1)*L)+columna;
    }
    /* Control para la busqueda del vecino Norte por parte de la fila supe-
rior */
    if (fila == L-1){
        /* Parte inferior de la red */
```

```
        *vecinoNorte = columna;
    } else {
        /* Control para la busqueda del vecino Norte por parte de cual-
quier fila */
        *vecinoNorte = ((fila+1)*L)+columna;
    }
    /* Control para la busqueda del vecino Oeste por parte de la columna iz-
quierda */
    if (columna == 0){
        *vecinoOeste = (fila*L)+(L-1);
    } else {
        /* Control para la busqueda del vecino Oeste por parte de cualquier co-
lumna */
        *vecinoOeste = (fila*L)+(columna-1);
    }
    /* Control para la busqueda del vecino Este por parte de la columna dere-
cha */
    if (columna == L-1){
        *vecinoEste = (fila*L);
    } else {
        /* Control para la busqueda del vecino Este por parte de cualquier co-
lumna */
        *vecinoEste = (fila*L)+(columna+1);
    }
}

/* Funcion para obtener el valor minimo de toda la red */
double obtenerMinimo(int rank, int vecinoNorte, int vecinoSur, int ve-
cinoEste, int vecinoOeste, double bufNumeros){
    /* Menor numero de toda la red iniciado al valor maximo que puede almace-
nar un double */
    double minimo = DBL_MAX;

    MPI_Status status;
    /* Calculo del valor minimo por filas, enviando al vecino Este el valor al-
macenado en cada iteracion */
    for(int i = 0; i<L; i++){
        /* Control para actualizar el valor que almacena cada nodo, quedan-
dose siempre con el de menor valor */
        if (bufNumeros<minimo){
            minimo = bufNumeros;
        }
        /* Envio del menor valor al vecino Este */
        MPI_Send(&minimo, 1, MPI_DOUBLE, vecinoEste, i, MPI_COMM_WORLD);
        /* Recepcion del menor valor por parte del vecino Oeste */
        MPI_Recv(&bufNumeros, 1, MPI_DOUBLE, veci-
noOeste, i, MPI_COMM_WORLD, &status);
        /* Control para actualizar el valor que almacena cada nodo, quedan-
dose siempre con el de menor valor */
```

```
        if(bufNumeros<minimo){
            minimo = bufNumeros;
        }
    }
    /* Calculo del valor minimo por columnas, enviando al vecino Norte el va-
lor almacenado en cada iteracion */
    for(int i = 0; i<L; i++){
        /* Envio del menor valor al vecino Norte */
        MPI_Send(&minimo, 1, MPI_DOUBLE, vecinoNorte, i, MPI_COMM_WORLD);
        /* Recepcion del menor valor por parte del vecino Sur */
        MPI_Recv(&bufNumeros, 1, MPI_DOUBLE, vecino-
Sur, i, MPI_COMM_WORLD, &status);
        /* Control para actualizar el valor que almacena cada nodo, quedan-
dose siempre con el de menor valor */
        if (bufNumeros<minimo){
            minimo = bufNumeros;
        }
    }

    return minimo;
}
```

4.3. HIPERCUBO.C.

```
/* OBTENCION DEL VALOR MAXIMO DE LOS DATOS DISTRIBUIDOS ENTRE LOS NO-
DOS DE UNA RED HIPERCUBO */
/* Carlos Gomez Fernandez */

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <string.h>
#include <math.h>

/* Archivo donde se almacenan los numeros a procesar */
#define FICHERO "datos.dat"

/* Funciones empleadas para la resolucion del problema */
int obtenerNumeros(double *numeros);
void obtenerVecinos(int rank, int *vecinos);
double obtenerMaximo(int rank, int *vecinos, double bufNumeros);

int main(int argc, char *argv[]){
    int rank, size, numerosProcesar, control = 1, procesos = pow(2,L), veci-
nos[L];
    double bufNumeros, maximo;
    MPI_Status status;
```

```
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    /* Control para que el rank 0 sea el que lleve a cabo el procesa-
miento y distribucion de los datos */
    if (rank == 0){
        /* Control del lanzamiento de procesos suficientes, acorde con el ta-
mano definido de la red */
        if (procesos != size){
            printf("[ERROR] Para un hipercubo de %d dimensiones deben ser lan-
zados un total %d procesos\n", L, procesos);
            /* Finalizacion de la ejecucion de los procesos */
            control = 0;
            MPI_Bcast(&control, 1, MPI_INT, 0, MPI_COMM_WORLD);
        } else {
            /* Vector de numeros a repartir entre los nodos de la red */
            double *numeros = malloc(1024 * sizeof(double));
            /* Obtencion de los numeros almacenados en el archivo datos.dat */
            numerosProcesar = obtenerNumeros(numeros);
            /* Control de la cantidad de numeros almacenados en el archivo da-
tos.dat, acorde con el tamano definido de la red */
            if(procesos != numerosProcesar){
                printf("[ERROR] Este hipercubo necesita únicamente %d nume-
ros y dispone de %d numeros en el fichero\n", procesos, numerosProcesar);
                /* Finalizacion de la ejecucion de los procesos */
                control = 0;
                MPI_Bcast(&control, 1, MPI_INT, 0, MPI_COMM_WORLD);
            } else {
                /* Continuar con la ejecucion de los procesos */
                MPI_Bcast(&control, 1, MPI_INT, 0, MPI_COMM_WORLD);
                /* Envio de los numeros a cada uno de los nodos de la red */
                for(int i = 0; i<numerosProcesar; i++){
                    bufNumeros = numeros[i];
                    MPI_Send(&bufNumeros, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
                }
            }
        }
    }
    /* El rank 0 envia un mensaje para continuar con la ejecucion de los proce-
sos, tras el envio de los numeros a cada uno de los nodos de la red */
    MPI_Bcast(&control, 1, MPI_INT, 0, MPI_COMM_WORLD);
    /* Control para la continuacion de la ejecucion de los procesos */
    if (control != 0){
        /* Recepcion de todos los nodos de los numeros envia-
dos por el rank 0 */
        MPI_Recv(&bufNumeros, 1, MPI_DOU-
BLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        /* Cada nodo obtiene sus vecinos */
```

```
        obtenerVecinos(rank, vecinos);
        /* Obtencion del valor maximo almacenado en la red */
        maximo = obtenerMaximo(rank, vecinos, bufNumeros);
        /* Control para que el rank 0 sea el que muestre el valor ma-
ximo de la red */
        if (rank == 0){
            printf("El mayor numero que almacena este hipercubo es %.1f\n", ma-
ximo);
        }
    }
    /* Finalizacion de la ejecucion */
    MPI_Finalize();
    return 0;
}

/* Funcion para obtener los numeros almacenados en el archivo datos.dat */
int obtenerNumeros(double *numeros){
    /* Vector auxiliar de char para almacenar los numeros separados por co-
mas */
    char *numerosFichero = malloc(1024 * sizeof(char));
    int *numeroActual;
    int totalNumeros = 0;

    /* Apertura del archivo en modo lectura */
    FILE *archivo = fopen(FICHERO, "r");
    /* Control de la correcta apertura del archivo */
    if (archivo == NULL){
        fprintf(stderr, "[ERROR] Error en la apertura del archivo\n");
        return 0;
    }
    /* Copia de los numeros en un vector auxiliar */
    fscanf(archivo, "%s", numerosFichero);
    /* Cierre del archivo */
    fclose(archivo);
    /* Lectura del primer numero entre comas mediante strtok y transforma-
cion a double con atof */
    numeros[totalNumeros++] = atof(strtok(numerosFichero,","));
    /* Lectura de los numeros entre comas*/
    while ((numeroActual = strtok(NULL,",")) != NULL){
        numeros[totalNumeros++] = atof(numeroActual);
    }

    return totalNumeros;
}

/* Funcion para obtener los vecinos de cada nodo */
void obtenerVecinos(int rank, int *vecinos){
    /* Todos los nodos tienen tantos vecinos como dimensiones tiene la red */
    for (int i = 0; i<L; i++){
```

```
        /* XOR entre el rank y la dimension de la red y almacena-
miento de los vecinos en el vector */
        vecinos[i] = (rank^((int)pow(2,i)));
    }
}

/* Funcion para obtener el valor maximo de toda la red */
double obtenerMaximo(int rank, int *vecinos, double bufNumeros){
    /* Mayor numero de toda la red iniciado al valor minimo que puede almace-
nar un double */
    double maximo = DBL_MIN;

    MPI_Status status;
    /* Calculo del valor maximo entre los vecinos de cada nodo, enviando a to-
dos los vecinos el valor almacenado en cada iteracion */
    for(int i = 0; i<L; i++){
        /* Control para actualizar el valor que almacena cada nodo, quedan-
dose siempre con el de mayor valor */
        if (bufNumeros>maximo){
            maximo = bufNumeros;
        }
        /* Envio del mayor valor a todos los vecinos del nodo */
        MPI_Send(&maximo, 1, MPI_DOUBLE, vecinos[i], i, MPI_COMM_WORLD);
        /* Recepcion del mayor valor por parte de todos los vecinos del nodo */
        MPI_Recv(&bufNumeros, 1, MPI_DOUBLE, veci-
nos[i], i, MPI_COMM_WORLD, &status);
        /* Control para actualizar el valor que almacena cada nodo, quedan-
dose siempre con el de mayor valor */
        if(bufNumeros>maximo){
            maximo = bufNumeros;
        }
    }
    return maximo;
}
```

5. INSTRUCCIONES COMPILACIÓN Y EJECUCIÓN.

Para llevar a cabo la compilación y ejecución de los programas se dispone de un archivo *Makefile*, además de un programa anexo denominado *obtenerNumeros.c*, el cual se encarga de la creación del archivo *datos.dat* necesario para la ejecución de los programas.

Para compilar y ejecutar el programa *obtenerNumeros.c*, necesario para la ejecución de la red Toroide, basta con ejecutar el siguiente comando en el directorio donde se encuentren almacenados tanto el archivo fuente, como el *Makefile*.
$ make obtenerNumerosToroide

Para compilar y ejecutar el programa *obtenerNumeros.c*, necesario para la ejecución de la red Hipercubo, basta con ejecutar el siguiente comando en el directorio donde se encuentren almacenados tanto el archivo fuente, como el *Makefile*.
$ make obtenerNumerosHipercubo

Para compilar el programa *Toroide.c*, basta con ejecutar el siguiente comando en el directorio donde se encuentren almacenados tanto el archivo fuente, como el *Makefile*.
$ make compilarToroide

Para compilar el programa *Hipercubo.c*, basta con ejecutar el siguiente comando en el directorio donde se encuentren almacenados tanto el archivo fuente, como el *Makefile*.
$ make compilarHipercubo

Para ejecutar el programa *Toroide.c*, basta con ejecutar el siguiente comando en el directorio donde se encuentren almacenados tanto el archivo fuente, como el *Makefile*.
$ make ejecutarToroide

Para ejecutar el programa *Hipercubo.c*, basta con ejecutar el siguiente comando en el directorio donde se encuentren almacenados tanto el archivo fuente, como el *Makefile*.
$ make ejecutarHipercubo

Para eliminar los archivos objeto generados en la compilación de los programas *obtenerNumeros.c* y *Toroide.c* basta con ejecutar el siguiente comando en el directorio donde se encuentren almacenados los archivos fuente, los archivos objeto y el *Makefile*.
$ make limpiarDirectoriosToroide

Para eliminar los archivos objeto generados en la compilación de los programas *obtenerNumeros.c* e *Hipercubo.c* basta con ejecutar el siguiente comando en el directorio donde se encuentren almacenados los archivos fuente, los archivos objeto y el *Makefile*.
$ make limpiarDirectoriosHipercubo

Para compilar y ejecutar tanto el programa *obtenerNumeros.c*, como el programa *Toroide.c* y eliminar posteriormente los archivos objeto generados, basta con ejecutar el siguiente comando en el directorio donde se encuentren almacenados los archivos fuente, los archivos objeto y el *Makefile*.
$ make ToroideCompleto

Para compilar y ejecutar tanto el programa *obtenerNumeros.c*, como el programa *Hipercubo.c* y eliminar posteriormente los archivos objeto generados, basta con ejecutar el siguiente comando en el directorio donde se encuentren almacenados los archivos fuente, los archivos objeto y el *Makefile*.
$ make HipercuboCompleto

6. Conclusiones

Resulta interesante destacar como, con la interfaz de paso de mensajes y con la inclusión mínima de primitivas *MPI*, se ha conseguido resolver un problema complejo, desarrollando un programa distribuido y altamente escalable. En cuanto a la complejidad del programa cabe destacar la obtención de los vecinos tanto en la red Toroide, como en la red Hipercubo, además de la comunicación entre vecinos de ambas redes para obtener la solución a cada uno de los problemas.